Implementing Hybrid Simulations that Integrate DES+SD in Python

Geraint I. Palmer[a] and Yawen Tian[a]

School of Mathematics, Cardiff University

ARTICLE HISTORY
Compiled November 10, 2021

ABSTRACT
Implementation of hybrid simulation is not well documented, and often obscured behind commercial software packages. Here two implementation approaches for integrating discrete event simulation and systems dynamics is presented in an open, reproducible framework, built with the Ciw library in Python. These two implementation approaches, denoted here by $DES \subset SD$ and $SD \subset DES$, correspond to two different process environments for DES+SD hybrid simulations. The models contribute to a collection of accessible and transparent hybrid simulations to be reproduced, scrutinised, built upon, and further developed. This paper also presents discussion on the lessons learned from these implementations, including the appropriateness of the models for different process environments, their ability to capture the full variability of a system, and subtleties in their synchronicity.

KEYWORDS
hybrid simulation; open source; implementation; discrete event simulation; systems dynamics

1. Introduction & Background

Computer simulation modelling is a useful methodology in a number of situations, from modelling complex systems, to communicating ideas, to validating analytical models. In the context of operational research, simulation modelling has traditionally been categorised into three main paradigms - discrete event simulation (DES), systems dynamics (SD), and agent based modelling (ABM). With the exception of ABM, these paradigms have clear boundaries in terms of context and use, as well as well-established processes and algorithms for their implementation. Hybrid simulations, that combine two or more of these paradigms, although not a new idea, are gaining popularity.

Hybrid simulations are primarily categorised by the nature of their component parts. Another common way to categorise hybrid simulations, used in Barbosa and Azevedo (2017); Brailsford et al. (2019); Morgan et al. (2017) and Swinerd and McNaught (2012), is to focus on the nature of the hybridisation: *comparison* in which separate models of the same system are compared; *sequential* in which one component it run and then another; *enriching* in which one primary component receives some information or insight from another minor component; *interaction* in which two equally important components model distinct aspects with clear boundaries, exchange information at runtime; and *integrated* in which both components are implemented together as

CONTACT Geraint I. Palmer. Email: palmergi1@cardiff.ac.uk

one whole approach, however to the authors' knowledge no precise definition of *integrated* has been given, for example in Brailsford et al. (2019) where the term is mentioned.

Recent reviews of its theory and use are given in Barbosa and Azevedo (2017); Brailsford et al. (2019); Sadsad et al. (2014); Scheidegger et al. (2018) and Tolk et al. (2018). However in the most comprehensive of these, Brailsford et al. (2019), it is noted that most papers focus heavily on the problems themselves and the conceptual model, with very little discussion or evidence on the implementation. One reason for this may be that when modelling a real system, the main focus should be the conceptual model, as we will discuss in Section 5, hybrid simulation for the sake of hybrid simulation may be inappropriate. However, this lack of details around hybrid simulation logic and implementation methodology is problematic for modellers wanting to understand, analyse, reproduce, extend, adapt existing models, or implement their own models. Furthermore, most models were implemented in closed source commercial software packages, and models were not shared or restrictive licensing inhibits their accessibility. This can lead to a number of problems: modellers have no access to examples of best practice and cannot scrutinise or duplicate implementation details, meaning modellers have to 'reinvent the wheel' each time they approach hybrid modelling. This reproducibility problem has also been discussed in non-hybrid simulation modelling, for example in Monks et al. (2019); Palmer et al. (2019) and Uhrmacher et al. (2016). This lack of reported details on implementation can also lead to mis-categorisations of hybrid models - the authors of Brailsford et al. (2019) state explicitly that they consider any model built in AnyLogic to be an example of 'interaction', despite not investigating the model implementation or code itself. This erroneous assumption that the software used for implementation is synonymous with the methodology itself, along with the shortage of published implementation details, may indicate that the words "hybrid simulation" may in fact mean different things and is not well defined.

In this paper we aim to contribute to the library of examples of implementation of integrated DES+SD hybrid simulation by providing reproducible examples of two approaches built in Python. This extends the work presented in Palmer et al. (2019); where the Ciw library was introduced for flexible object-orientated reproducible DES. In this paper we demonstrate that library's applicability to DES+SD hybrid simulation through two exemplars that contribute to the collection of open and accessible DES+SD hybrid models. Implementation details are discussed explicitly, facilitating their reproduction, modification and scrutinisation.

Python has a permissive license and can be used without cost, and all models discussed in this paper are archived and available in their entirety for scrutinisation and modification at Palmer and Tian (2021). Specifically we combine Ciw with the SciPy library for numerically solving the differential equations of the SD components. The authors have strived to use best practices in research software development (Benureau and Rougier (2018); Percival (2014); Wilson et al. (2014)) such as modularisation, documentation and readability, accessibility, and testing (which can contribute to simulation model verification). Section 2 gives implementation details in a software-agnostic manner. These form the principles underlying the Python implementation discussed in Section 3. Examples of both approaches are given in Section 4, and finally Section 5 discusses what we have learned about DES+SD hybrid simulation from these implementations. Throughout this paper DES will mean discrete event simulations of queueing networks.

2. Methodology

The key concept behind both approaches is that the continuous SD aspects are solved in between each discrete event in the DES aspect. The resulting state of the SD components after the latest update effects how the discrete events are carried out or scheduled, while the resulting state of the DES components change some parameters of the SD models. This is a small adaptation to

the event scheduling DES approach described in Robinson (2014), this is shown in Figure 1, where the **A**-phase corresponds to advancing the clock, the **B**-phase corresponds to scheduled events, and the **C**-phase corresponds to unscheduled events.

In this section some aspects of the hybrid simulation methodology will be discussed.

2.1. The process environment

Hybrid simulation involves combining two fundamentally different views, the continuous macroscopic homogeneous strategic and deterministic perspective of SD, and the discrete detailed heterogeneous operational and stochastic perspective of DES. A comprehensive comparison of these and some early ideas for their integration is given in Brito et al. (2011).

Methodologies that combine these views have been categorised in a number of ways: by the nature of their components (DES+SD, DES+ABM, SD+ABM, or DES+SD+ABM), by the nature of their interaction (comparison, sequential, enriching, interaction, or integration) (Barbosa and Azevedo (2017); Brailsford et al. (2019); Morgan et al. (2017); Swinerd and McNaught (2012)), by the level of automation of information transfer (automated integration, manual integration, or integration using intermediate tools) (Brailsford et al. (2019)), and by how the components connect in the conceptual model (hierarchical, process environment, or integrated format) (Chahal and Eldabi (2008)). These categorisations are not necessarily independent from one another. For example process environments where continuous and discrete components influence one another synchronously would be impossible to implement in a sequential manner due to the feedback loops between the components; and a fully integrated model would be practically infeasible if data were to be manually transferred between components.

Here we consider integrated DES+SD models using automated information transfer within the same software. We will consider two different approaches, corresponding to two ways of thinking of the process environment. In one approach, which we will refer to as DES $\subset$ SD, one or more DES models live inside an environment modelled by SD. In another, which we will refer to as SD $\subset$ DES, one or more components modelled by SD live inside an environment defined by DES. Figure 2 shows a diagram of these approaches. Here solid arrows indicate interactions that *must* occur, while dashed arrows indicate optional interactions. Examples of both approaches implemented in detail are given in Section 4.

Note that it is important that the solid arrows are bidirectional, that is in integrated DES $\subset$ SD the DES component must influence the SD environment, and the SD environment must influence the DES component. If these are unidirectional, then a sequential hybrid simulation is sufficient.

Note also that neither approach, DES $\subset$ SD nor SD $\subset$ DES, imply that one component is more dominant than the other. The amount of influence each component has on the system is determined by the problem itself, and not the implementation methodology.

2.2. Time points

SD simulations contain stocks, flows between them, and other influencing variables. Running an SD simulation usually involves numerically integrating a set of differential equations that describe the relationships between these elements. This involves splitting the time domain into many discrete time steps and implementing some Euler or Runge-Kutta methods (Pidd (2004), Seeler (2014)), which are often abstracted away in specific commercial packages.

In these approaches, the entire time domain of the simulation, from time 0 until the end of the simulation, is split into time slices Δt. Now let $T^\star = \{0, \Delta t, 2\Delta t, 3\Delta t, ...\}$ be the base set of

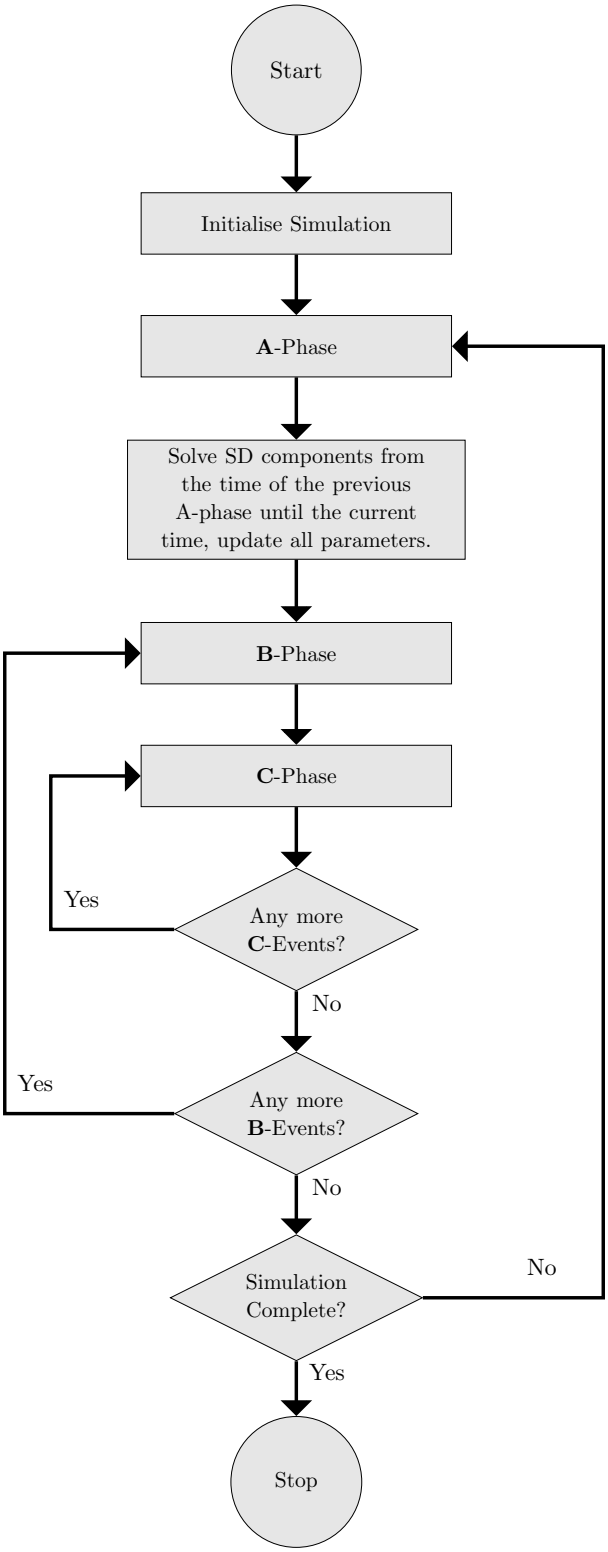
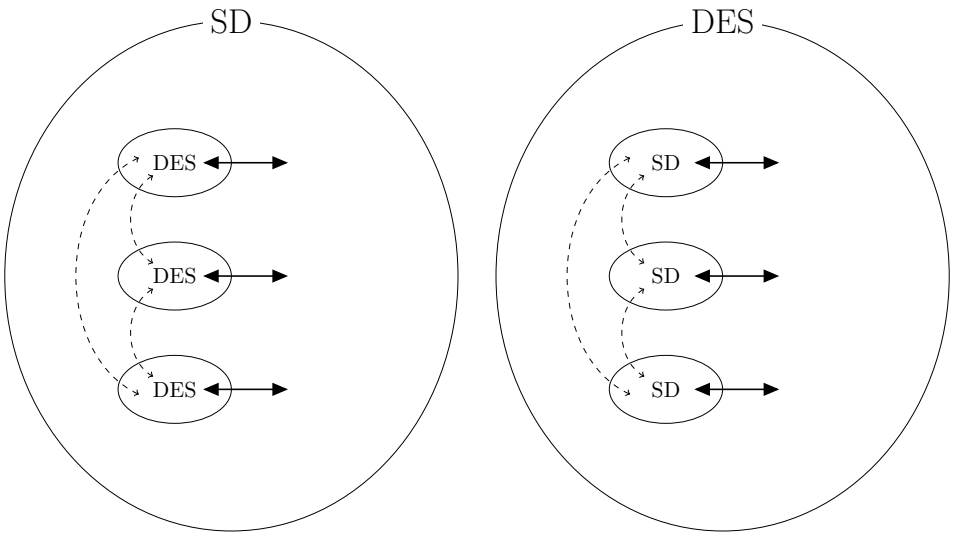


Figure 1. Hybrid SD adaptation to the event scheduling approach. Diagram adapted from Robinson (2014).



(a) DES ⊂ SD (b) SD ⊂ DES

Figure 2. Overview of the process environments for two approaches to DES+SD hybrid simulation.

time points to use to numerically integrate the SD components over, when no discrete events are scheduled. This is shown in the first time-line in Figure 3. Note that these need not necessarily to be equidistant.

Now when we consider hybrid simulations, this SD component happens at the same time as the DES components. In these components events will occur at times points separated by time intervals that are usually but not necessarily random, and won't necessarily coincide with the elements of $T^\star$. Only some of these events have the potential to affect or be affected by the parameters of the SD component, so only those are considered here. Let $T_1 = \{t_{1,1}, t_{1,2}, t_{1,3}, ...\}$ be the set of times in which these concerned events occur. This is shown in the second time-line in Figure 3. Note that, as unscheduled **C**-events occur immediately after, but at the same simulation time as scheduled **B**-events, then all time points in T_1 will correspond to scheduled **B**-events.

Now once the DES component reaches a scheduled event, before that event is carried out, the SD component is numerically integrated across relevant time points between the time of the previous event and the current time inclusive. In this case, once the DES component reaches the event at time $t_{1,4}$, the SD component is numerically integrated across $\{t_{1,3}, 13\Delta t, 14\Delta t, 15\Delta t, 16\Delta t, 17\Delta t, 18\Delta t, t_{1,4}\}$. Therefore the resulting stock levels of the SD component over time would be a continuous series of piecewise curves with subinterval endpoints the elements of T_1.

2.3. Synchronicity

A vital consideration when implementing hybrid simulation models is synchronicity (Brailsford et al. (2019)), that is ensuring that the time advancement mechanism of all model components do not violate causality. In the approaches outlined here, as shown in Figure 1, at each event:

- the clock is advanced to the current event,
- we go 'back in time' and numerically integrate the SD component from the previous event until the current time, using DES parameters resulting from that previous event,
- using updated parameters resulting from the SD component, the current event is carried

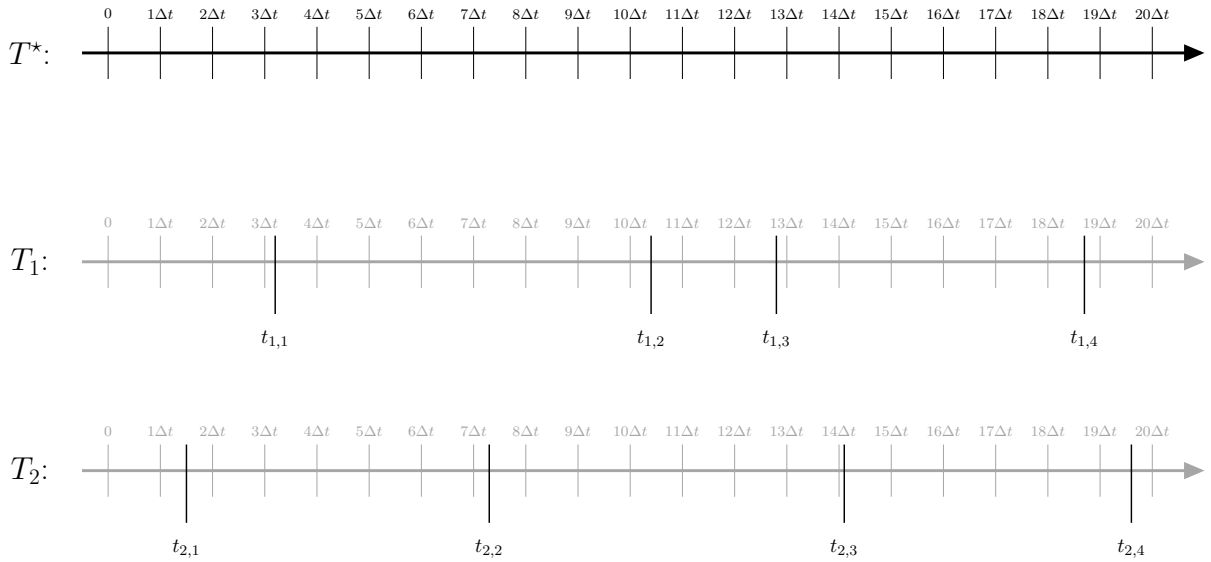


Figure 3. Example of the meaningful time points of the DES+SD hybrid model.

out.

Thus each time the DES components carry out an event, the SD components are up to date. Meanwhile, for each time period the SD components are solved, the DES components are up to date. This ensures that no new synchronicity issues arise from the integration of both component types.

However, from the point of view of the DES components, these hybrid models are simply a DES with time-dependant parameters (these parameters changing as the SD component changes). Therefore all the usual synchronicity issues with time-dependent DES models still need to be considered (Pidd (2004)). The main issues to consider are events that need to be scheduled in advance, but requiring future parameters, for example scheduling arrivals from an inter-arrival distribution.

For example, consider a queue in which arrivals occur on average once a day Monday to Friday, but on average once an hour Saturday and Sunday. If an arrival event occurs late on Friday, then the next arrival will be sampled using Friday's inter-arrival distribution, and so may not be scheduled until Saturday afternoon. However Saturday morning is modelled as having arrivals on average once an hour, but the simulation never stopped on Saturday morning to sample any arrivals. When using DES in isolation this may be overcome by re-sampling on Saturday morning, the discrete time point that the inter-arrival distribution changes.

However, this is not possible if the parameters of the inter-arrival distribution change continuously, which may be the case if using the continuously changing stock levels of the SD components - changes that we are only aware have occurred once they have already happened. It would also be incorrect to repeatedly re-sample the next arrival date at regular intervals, as we would in effect be taking the minimum of a number of random variables, which has a different distribution to that required. Therefore care should be taken if the hybrid model can potentially contain large and sudden changes in future samples parameters, for example arrival rates.

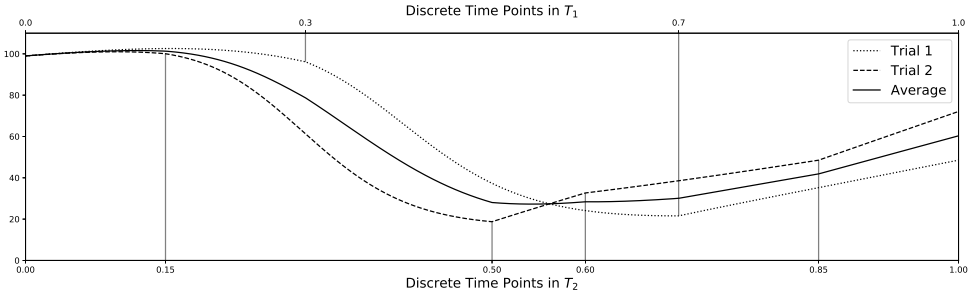


Figure 4. Illustrative example of taking averaging stock levels over trials. Here $T_1 = \{0.3, 0.7\}$, and $T_2 = \{0.15, 0.5, 0.6, 0.85\}$.

2.4. Accounting for variability

One particular problem to overcome with hybrid simulation, which arose when running the exemplar in Section 4.1, is accounting for the variability in stock levels over time. Usually SD models consider only expected values in stock levels over time, and therefore it is not usual to run several trials of an SD; however in DES+SD hybrid simulations the DES components may still bring inherent randomness to the model results. Therefore taking averages over trials is still required. This is due to the stochastic DES component feeding in different stochastically derived parameters to the SD component, and having stochastic inter-event times which determine when the stochastic endpoints of piecewise SD components.

However, using the technique described above, each trial will have a different set of event times, as shown in the third time-line in Figure 3. That is T_1 and T_2 have different elements. Therefore after the first trial the stock levels of the SD components will have values for all elements in $T^\star \cup T_1$, and after the second trial they will have values for all elements in $T^\star \cup T_2$.

Thus we propose the following: in order to take the average stock level across all possible time points, stock values for each trial are required for all elements in $T^\star \cup T_1 \cup T_2$. Clearly the first trial have no values for elements in $T_2 \setminus T_1$, and the second trial have no elements for $T_1 \setminus T_2$. Therefore values for these time points would need to be interpolated from the closest available values either side of the missing point. This can be generalised to any number of trials. Interpolation is important here to avoid sudden jumps, as in random systems it would be very unlikely that any of the T_i's would share any elements.

This method allows us to smooth over the variability produced by the DES components. An illustrative example of taking an average over stock levels from two trials of a DES+SD hybrid system is shown in Figure 4.

Additionally, it is important to note that SD methodology does not usually explicitly model variability, with stock levels representing expected values only. Therefore any variability produced by the hybrid model will only come from the DES component, and so may not represent the full variability in the system. It is vital that this is kept in mind whenever KPIs of hybrid simulations are collected, especially if considering their variability such as their confidence intervals. One potential route to overcome this is by formulating and numerically solving the SD component as stochastic differential equations (Kloeden and Pearson (1977)).

3. Implementation

The approaches described here make use of two Python libraries that can be used to implement DES and SD simulations respectively, Ciw (Palmer et al. (2019); The Ciw library developers

(2020)) and SciPy (Virtanen et al. (2020)). Ciw builds and runs DES of queueing systems in an object-orientated manner, while SciPy has functions for numerically integrating systems of differential equations of the SD components. A key feature is SciPy's use of the Livermore Solver for Ordinary Differential Equations (Radhakrishnan and Hindmarsh (1993)), which allows for numerically integrating systems of differential equations over time points that are not equidistant using an intelligent choice of techniques. The use of both libraries is detailed extensively in their respective documentations.

The question then, is how to integrate them? Ciw's open and object orientated nature allows ease of extension and modification. This is done through the concept of inheritance (Ramalho (2015)). In Python this means creating new classes of objects from other classes, which inherit all attributes and methods of the parent class, but allows from some modification. Therefore one could create new hybrid simulation classes by inheriting from Ciw's collection of classes specialised for DES, and adapting them to have hybrid functionality. This would mean that most of the code would not have to be rewritten, as all methods and attributes from the original classes would be present in the inherited classes, through inheritance. Only the new or adapted behaviour needs to be explicitly written.

Throughout this paper text in monospace font represent classes, objects and functions in the software. Relevant classes here include the `Simulation` that holds all other objects and runs the whole simulation; `Individuals` that are sent around a network of `Node`s and are served by `Servers`; the `ArrivalNode` that spawns new individuals; the `ExitNode` that collects completed individuals; and a number of `Distribution` classes that sample inter-arrival, batching, and service times. These connect together to form a DES as shown by the white ellipses in Figure 5.

In order to build hybrid simulations, we propose creating a new objects containing the SD stocks, parameters, and differential equations, along with a method to numerically integrate them over the specified time domain. These objects will be connected to the main Ciw framework in different ways, depending on the the approach (DES $\subset$ SD or SD $\subset$ DES) and the conceptual model:

- In DES $\subset$ SD there are a number of DES components within one SD environment. In general multiple DES components can be implemented as just one DES using a shared event scheduling approach (in Ciw this would correspond to multiple disconnected components of the same queueing network), and so there would only be one `Simulation` object. The single SD object would then connect to this `Simulation` object, where all other objects have indirect access to it.
- In SD $\subset$ DES there are a number of SD components within one DES environment. Depending on the conceptual model, each of these SD components may lie within the `Individual`, `Server` or even `Node` objects.

These are shown visually in Figure 5.

In order to 'link' these objects together we make use of inheritance to modify the existing Ciw objects. Two modifications are required: giving the relevant Ciw objects an attribute that points to the SD component object, and modifying any event methods so that they numerically integrate the SD systems and make use of their updated parameters.

Details of these modifications for specific models are given in Section 4.

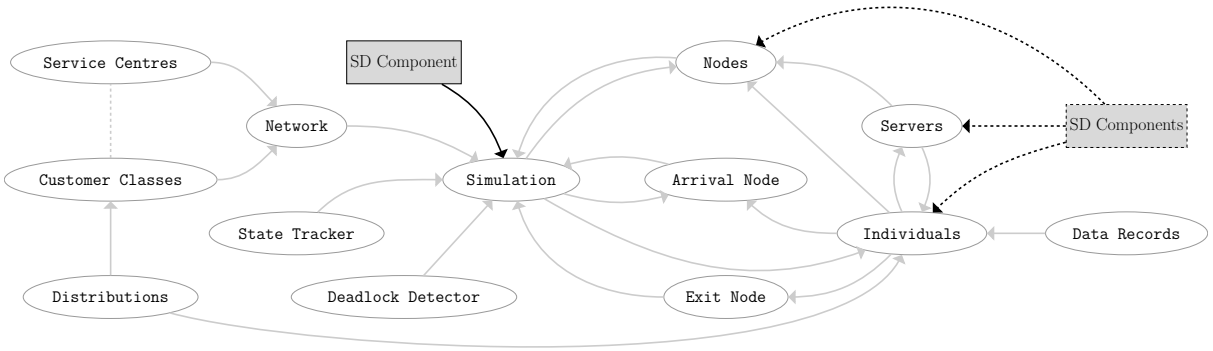


Figure 5. Ciw's code structure and where the SD components would sit. White ellipses represent objects that are part of the Ciw library; the solid shaded rectangles represents where the SD component would sit in the DES $\subset$ SD approach, and the dashed shaded rectangle represents where the SD components would sit in the SD $\subset$ DES approach. Adapted from Palmer (2020).

4. Illustrative Examples

In this sections two illustrative examples of given, one of DES $\subset$ SD, and one of SD $\subset$ DES. All scenarios and parameters are fabricated in order to demonstrate the methodology and implementation. All models are archived and publicly available at Palmer and Tian (2021), and the code is also included as Appendices. Throughout this section references will be made to specific files and lines of code in those Appendices. All models were also built in AnyLogic, also archived at the same place, with results obtained with the AnyLogic models given in Appendix E for comparison.

4.1. The DES $\subset$ SD Model

4.1.1. Premise

Consider a small town living with some infectious disease, and under some form of lockdown similar to what occurred in the UK under Covid-19 at the end of March 2020. There is one supermarket, and individuals only leave their homes to go to the supermarket to buy essentials.

Individuals can be in one of four disease states, susceptible S, infected I, temporarily immune M, or dead D. These are influenced by the infectivity of the disease α, the contact rate c, the fatality rate f, the recovery rate r, the proportion of people recovering who gain a temporary immunity p, the rate at which they lose this immunity τ, the frequency individuals go to the supermarket γ, and the proportion of infected people still going to the supermarket either due to being asymptomatic or through negligence β. The relationships between these stocks and parameters are given in Equations (1) to (4).

$$\frac{dS}{dt} = -\frac{SI\alpha\beta c}{S+\beta I+M} + (1-p)\,rI + \tau M \tag{1}$$

$$\frac{dI}{dt} = \frac{SI\alpha\beta c}{S+\beta I+M} + (r+f)\,I \tag{2}$$

$$\frac{dM}{dt} = prI - \tau M \tag{3}$$

$$\frac{dD}{dt} = fI \tag{4}$$

As individuals only leave their homes to go to the supermarket, the contact rate c is determined by the number of people present in the supermarket L. It is the frequency that individuals go to the supermarket multiplied by the current number of other customers present, given by Equation (5), where L is to be determined from the DES component at any given time.

$$c = \gamma \max(L-1, 0) \tag{5}$$

Finally the arrivals to the supermarket are modelled as a Poisson distribution, with rate Λ. This rate is determined by the number of people in the town who are able to go to the supermarket, and the frequency that each person goes, given by Equation (6).

$$\Lambda = \gamma\,(S+\beta I+M) \tag{6}$$

The supermarket has a maximum customer capacity, and if customers arrive and the supermarket is at capacity, they must wait in a queue outside the shop. Here capacity is modelled as servers, and the queue to enter is modelled as a queueing buffer; time spent in the shop is the service time, and this includes the time spent waiting at the checkout inside the shop. This service time is modelled as an Exponential distribution with rate μ.

Although there are no standard ways to represent this system diagrammatically, it is fairly straight forward to combine stock and flow diagrams and queueing diagrams by connecting them with influence arrows typical of SD representations. A representation of this scenario is given in Figure 6.

4.1.2. Implementation

This scenario fits the DES $\subset$ SD approach, as there is one continuous component that can be modelled as SD affecting and being affected by the discrete queueing system. The code for this implementation is given in Appendix A. As described in Section 3, one SD component object is defined, `SD_Component` (lines 5-42), containing attributes holding the SD component parameters and methods to numerically solve the system. The DES component uses the Ciw framework with minor modifications in order to trigger the `SD_Component` to solve the system at the appropriate times. This requires a `HybridSimulation` object which inherits from Ciw's `Simulation` object, a `HybridNode` which inherits from Ciw's `Node` object, and a new inter-arrival distribution object, `SolveSDArrivals`. This relevant modifications are:

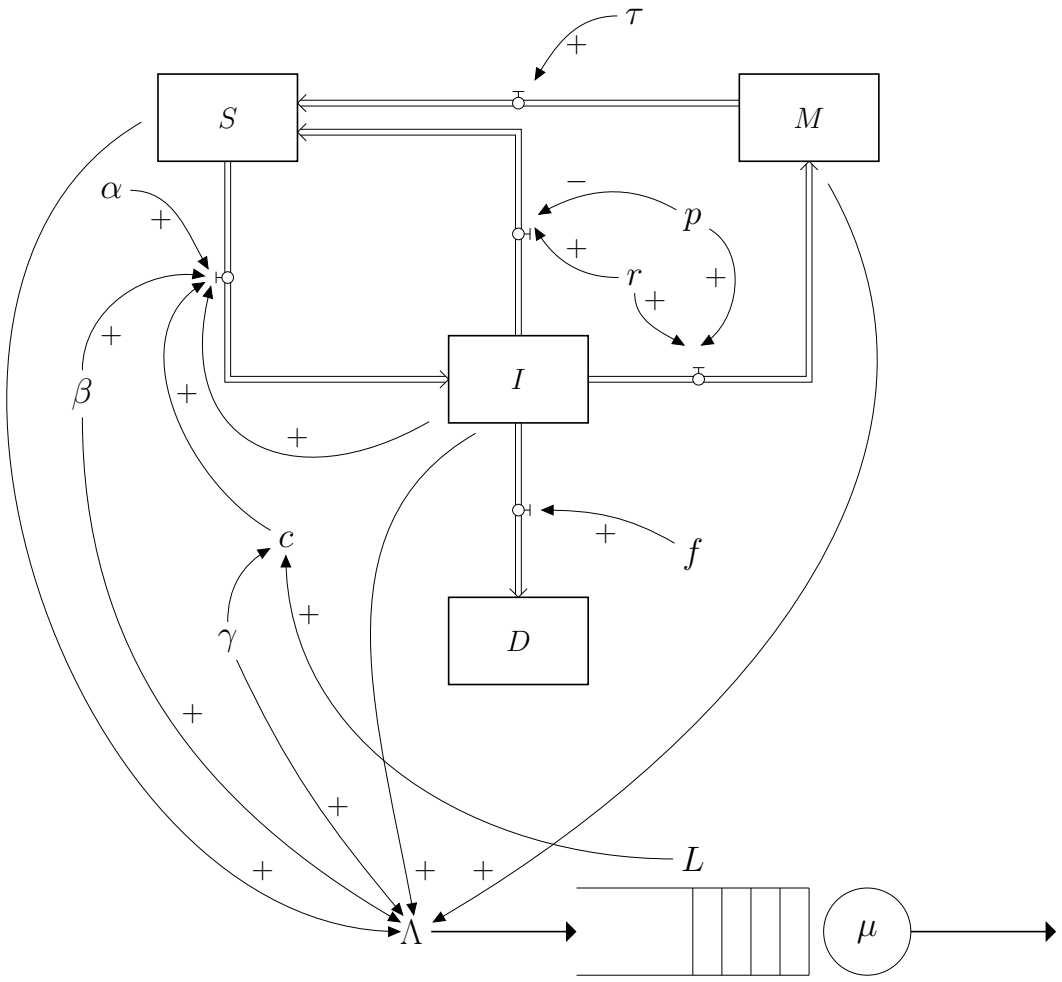


Figure 6. A combined stock-and-flow / queueing diagram for the supermarket during a pandemic.

Table 1. Parameters used in the example DES $\subset$ SD model of a supermarket in a pandemic.

$S_0 = 200$	$I_0 = 2$	$M_0 = 0$	$D_0 = 0$
$\alpha = 1$	$\beta = 1/3$	$r = 1/14$	$\gamma = 1/2$
$p = 1/4$	$\mu = 48$	$\tau = 1/75$	$f = 1/21$

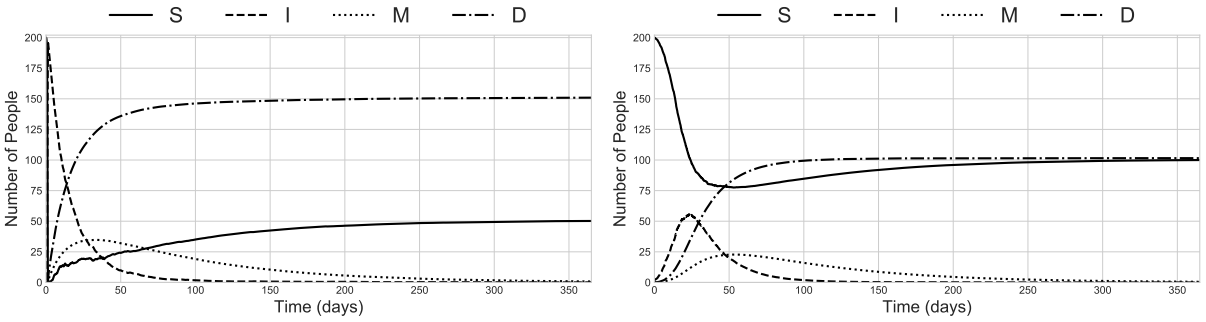
- The `HybridSimulation` object needs to be given the `SD_Component` object as an attribute, which needs to happen at the object initialisation stage (line 53);
- When the `HybridSimulation` starts to simulate, the `SD_Component` needs to know the time domain $T^\star$ (line 57);
- After the final event of the simulation, the `SD_Component` needs to solve SD system for the rest of $T^\star$ (line 59);
- Each time the number of individuals L in the system changes, the `SD_Component` needs to solve the SD system from the previous time L changed. This is done each time a new arrival occurs, and thus each time a new arrival is sampled (lines 63-69), and each time an individual leaves the system (line 48).

Once these new objects are defined with the above modifications, the system can be run over a given number of trials. The Python library Pandas (McKinney (2010)) is used for the interpolation and to take average stock levels.

4.1.3. Parameters & Results

A function to run a trial of this model and to collect KPIs is given in Appendix B. Table 4.1.3 gives the parameter values and initial stock levels used.

Two scenarios were run, the first allowing only one customer in the supermarket at a time while all other customers queued, and the second allowing in three customers at a time. Average stock levels across 10 trials, for one year of simulated time, with $T^\star$ taken to be 50 thousand equally spaced time points over 365 simulated days, are given in Figure 7. Note here that no warm-up time was used, this is discussed further in Section 5. This model shows that with a capacity of one the town would experience average total of 150.1 deaths, whereas with a capacity of three the town would only experience 101.5 deaths. This is due to limited capacities resulting in longer queues, and thus more contacts than would be seen without capacities.



(a) Maximum capacity of one.

(b) Maximum capacity of three.

Figure 7. Resulting stock levels for the DES $\subset$ SD model with supermarket capacities of one and three.

These scenarios were chosen to show that modifying some specifications on the operational level that are usually associated only with DES, in this case number of servers / capacity, can have profound effects on strategic KPIs usually associated only with SD, in this case expected

numbers of susceptible, infected and dead people over a year. This interaction between the operational, tactical, and strategic levels would be much more difficult to model using DES or SD in isolation. However, analysis of variability is weakened.

4.2. The SD ⊂ DES Model

4.2.1. Premise

Now an example of the second approach is given, where a number of entities modelled by SD interact with a larger system modelled by DES. Consider treatment for a degenerative disease such as cancer. As a patient waits for treatment they continue to degenerate, resulting in a worse condition by the time treatment begins and so longer treatment times.

Each patient contains a percentage of healthy cells H, and unhealthy cells U, such that $U + H = 100$. Each unhealthy cell converts healthy cells at a rate k per month. When receiving treatment the degenerative process continues, and the treatment reverses the degenerative process by c healthy cells per month. The relationships between these stocks and parameters are given in Equations 7 and 8, where $c = 0$ whenever the patient is not receiving treatment.

$$\frac{dH}{dt} = \frac{1}{100}(c - k)UH \tag{7}$$

$$\frac{dU}{dt} = \frac{1}{100}(k - c)UH \tag{8}$$

Treatment is modelled as a queueing system, the capacity modelled as servers and the waiting list modelled as the queueing buffer. Patients are referred for treatment at a rate Λ per month, with an initial H level sampled uniformly between 40 and 90. Patients are treated until their percentage of unhealthy cells reach 1%, their service time, s, can then be found by solving Equation (8) with the substitution $U = 100 - H$, with an initial condition being that patient's U stock level at the time treatment begins $t = 0$, this is given in Equation (9).

$$s = \frac{1}{k - c} \ln\left(\frac{H}{99U}\right) \tag{9}$$

Furthermore, if by the time a patient is due to begin treatment their U level is below 10%, then they are deemed untreatable and leave the system immediately. We are interested in finding a relationship between the capacity of the treatment programme and the number of patients deemed untreatable by the time they begin their treatment.

Again, there are no standard ways to diagrammatically represent SD+DES hybrid systems, and it is a little more difficult to represent a systems in which an indefinite amount of entities all contain an SD component. An attempt at a representation of this scenario is given in Figure 8; the dotted ellipse represents that the SD component within is applicable to each patient, and the dashed influence line represents that the parameter c changes depending on whether the patient is queueing or in service.

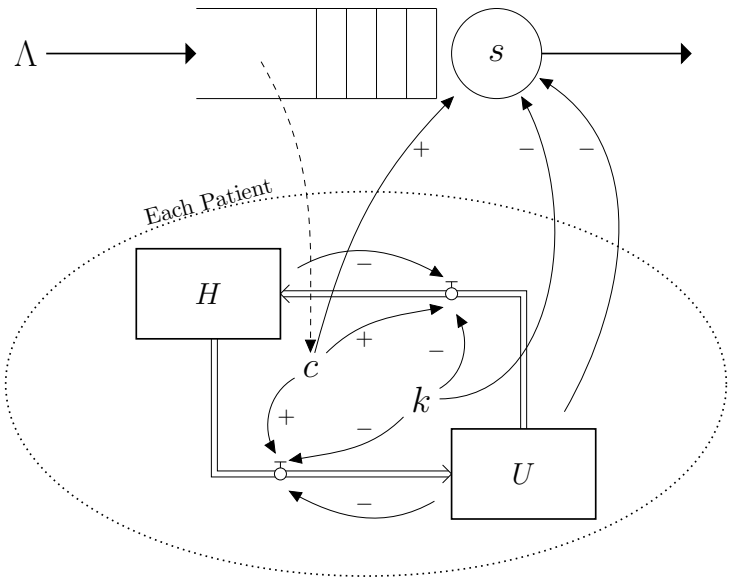


Figure 8. A combined stock-and-flow / queueing diagram for cancer patient treatment.

4.2.2. Implementation

This scenario fits the SD ⊂ DES approach, as there are many continuous components that can be modelled as SD affecting and being affected by the discrete queueing system. The code for this implementation is given in Appendix C. As described in Section 3 we can define SD component objects and attach them to each individual that enters the system, however Ciw already contains `Individual` objects that don't do much, so we can create a new `SD_Individual` object that inherits from this which contains the SD parameters and methods to these objects directly. We can then add all other necessary minor modifications to Ciw's other objects in order for the SD mechanisms to trigger at the correct times. In this case this requires a `HybridSimulation` object which inherits from Ciw's `Simulation` object, a `HybridNode` which inherits from Ciw's `Node` object, a `HybridArrivals` object that inherits from Ciw's `ArrivalNode` object, and a new service time distribution object, `Treatment`. This relevant modifications are:

- The `SD_Individual` object is defined. It needs to know the system parameters and initial stock values (lines 7-14), it needs a method containing the differential equations used to describe the SD component (lines 16-24), and a method to numerically solve these for a given time period (lines 26-39);
- The `HybridSimulation` needs to be given a global $T^\star$, which all individuals can refer to (line 68);
- The `Treatment` object needs to trigger the SD solve for their current individual and use the updated stock levels to sample a service time (lines 43-49);
- The `HybridNode` object need to trigger the SD solve for their current individual when they are released (line 55).

4.2.3. Parameters & Results

A function to run a trial of this model and to collect KPIs is given in Appendix D. The parameter values used are $k = 0.08$, $c = 0.4$, and $\Lambda = 2$. Twenty trials of the simulation were run for three years, with one year warm up to ensure that the models reach steady state, and one year cool down to ensure all completed records are collected. Models are run for all capacities between 20

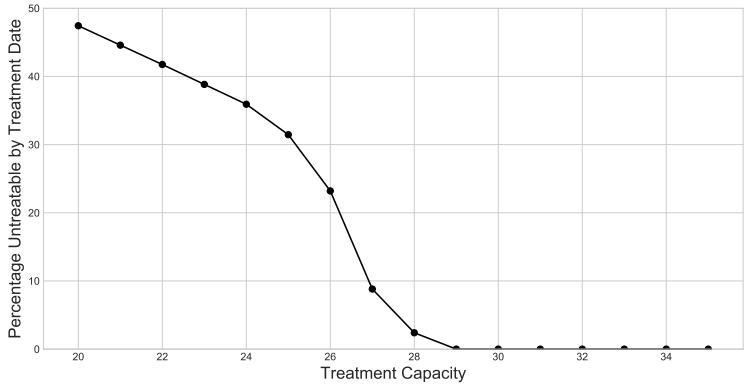


Figure 9. Proportion of patients untreatable ($U < 10\%$ at treatment start), for different treatment capacities.

and 35, the KPI of interest is the proportion of patients untreatable, which is collected as the average value over twenty trials. Figure 9 shows the results.

5. Discussion

In the previous section we showed that the methodologies described in Section 2 and their implementation described in Section 3 are feasible. This made use of inheritance to adapt classes built for DES in order to share information with and solve SD components. A similar technique could be utilised to adapt these classes so that objects interact with one another, or learn as the simulation progresses, and thus incorporating elements of ABM into DES modelling too.

This paper focussed on two types of process environment, referred to here as DES $\subset$ SD and SD $\subset$ DES. As shown in Figure 2, the process environment allows for a number of sub-components, as shown in the second exemplar in Section 4.2. Here each individual was itself an SD component, interacting bidirectionally with the DES environment. These individuals were however independent of one another, they did not directly interact with one another. There is potential for these direct interactions between components, as shown by the dotted arrows in Figure 2, due to the object orientated nature of the components described in this implementation; for example one customer's parameters may be affected by the attributes of the customers who they are sharing a queueing buffer with. This type of inter-component interaction may be considered a form of DES+SD+ABM.

With the Python implementation presented in this paper, a nested SD $\subset$ DES $\subset$ SD approach is also possible, that is each of the DES components in the DES $\subset$ SD may also act as environments where further components modelled as SD may live. However, the authors do not believe that a nested DES $\subset$ SD $\subset$ DES system is possible to implement in this framework, and cannot imagine a situation where this would be required.

During the building and running of two exemplars, one for each process environment type, a number of lessons were learned about the nature of hybrid DES+SD models built in this way.

One of the immediate apparent disadvantages of this method is that it is not immediately out-of-the-box ready such as commercial software. Understanding of Python, the Ciw library infrastructure, the event scheduling mechanism of the DES component, numerically solving differential equations, and the hybrid methodologies are all required to build a model. However, as

discussed in Palmer et al. (2019), this practice has numerous advantages such as reproducibility, transparency, and customisation. This is supported by the experiences of the authors in Brailsford et al. (2013) where they note that some users may find commercial software limiting, frustrating, and opaque, despite their apparent convenience, while a much more open environment offers more flexibility and transparency; and in Viana et al. (2014) where combining separate commercial software is noted as being challenging. The open, transparent, and accessible models presented in this paper help address this particular knowledge barrier.

Other advantages include that this style lends itself well to good verification with the inclusion of a well-written automated software testing suite (Percival (2014)), ensuring that the methodologies introduced in Section 2 is implemented without any bugs or unforeseen consequences. Furthermore, collaboration in implementation and model development is eased, as the models' scripted nature lends itself well to version control systems, which can clearly highlight changes without the need to hunt through GUI menus. As was originally noted in Palmer et al. (2019) for Ciw DES models, the hybrid models built in this paper still exist in a Python environment, allowing seamless integration between the simulation and both pre- and post- model data analysis, as well as allowing the hybrid models to be part of even larger methodologies such as optimisation algorithms.

Other considerations concern the appropriateness of hybrid methodologies to the problem at hand. That is, just because you can model something with hybrid simulation, doesn't mean you should.

The first of these considerations was mentioned in Section 2.4, that is that it's important to be aware that hybrid DES+SD models may not be fully capturing a system's variability as well as DES used in isolation. This is due to parameters calculated from SD components in general representing expected values only and not capturing variability. When using DES in isolation measuring the variability of KPIs are often meaningful, for example a confidence interval around the mean waiting time; or the KPI itself may be a measure of variability, for example the proportion of trials a certain event occurred in. We can usually be confident that DES is a valid approach for estimating these KPIs as most sources of variability are captured explicitly as inputs to the model. However SD modelling is concerned mainly with expected values, and any inherent variability in the system being modelled is not captured. Therefore in DES+SD hybrid simulations, only part of the system's inherent variability is being captured, and it may even be difficult to recognise how much of the variability is captured.

As with all models, the limitation of their components parts accrue when combined, therefore this may be serious limitation given that details about variability are often desired outputs for DES components. For example, in Figure 7, we attempted to calculate confidence intervals for the stock levels over time, based on the outcomes of the trials. However as variability in the SD components were not modelled, despite being inherent in the system, it was felt that these confidence intervals would in reality have much smaller confidence, though it was not possible to measure by how much.

Another consideration is the size and direction of the effects the DES and SD components have on one another. As mentioned briefly in Section 2.1, if the links between the DES and SD components are not bidirectional, then a sequential simulation will suffice. For example, if the DES component is affected by some larger SD environment, but the SD environment is not affected by what goes on in the DES component, then running the SD component alone will still be valid. The DES model is run over the same time domain, with the results of the SD model fed into it some time-dependent manner.

Even when the links are bidirectional, a hybrid model may be unnecessary if the effect sizes are not large enough to significantly effect each component. This became apparent when attempting

Table 2. The main *Input*, *Process*, and *Output* of the three main simulation paradigms.

	Input	*Process*	*Output*
DES	Processes and events.	Scheduling events and discrete jumps to event times.	Detailed individual statistics.
SD	Stocks, flows and influences.	Numerically integrating a system of differential equations.	Expected stock levels over time.
ABM	Interactions between agents.	Varied.	Emergent behaviour.

to choose parameter values for the two exemplars presented here, as many parameter sets considered sensible to the authors did not display demonstrable hybrid effects. Effect sizes and time domains of the component models would need to be sufficiently similar in order for hybrid modelling to become useful.

Clearly for the methodology presented in this paper to work both components need to be run over the same time domain. However if, for example, events of the DES component take place minutes apart, while any meaningful changes to the SD component take years, then the components are unlikely to significantly effect one another. In the case where the SD component changes much faster than events of the DES component are to be scheduled, then the model may introduce synchronicity issues, as discussed in Section 2.3. Similarly, even if changes in both components happen on the same time scale, if the effect sizes of the links are small, it may prove no different to running the models separately. Notwithstanding, running an integrated hybrid model may hold the key to discovering this lack of effect.

Note that it is the difference in the scales of these effect sizes and time domains that determine whether one component is more dominant than the other, not the process environment or the implementation, as noted in Section 2.1. Complete dominance, where the scales of both components are too different to affect one another, would not require hybrid modelling at all.

Given these potential limitations, when is this methodology appropriate?

Three important perspectives are discussed in Morgan et al. (2017): the model *input*, the model *process*, and the model *output*. Table 5 gives these for the three main simulation paradigms. Hybrid simulations may be appropriate if a model has multiple different equally important inputs corresponding to different simulation paradigms, or if the required output does not correspond with that of the paradigm of the most important input. Otherwise an attempt at hybrid simulation may well simplify to one of the standard well-established paradigms, for example a time-dependent DES, or a piecewise SD.

In addition to these conditions, as discussed above, the experience of developing the exemplars in this paper have highlighted some other factors to help determine the appropriateness of integrated models. In summary, these include the directionality of the component links, time scales and effect sizes, and importance of output variability.

6. Conclusions

This work follows on from a previous paper, Palmer et al. (2019), where it is noted that the open, modular, and modifiable nature of Ciw simulations, along with the Python ecosystem in which they sit, and Python's extensive range of scientific libraries, mean they are amenable to mixed-methods modelling. In this paper we demonstrate that this is not just possible in theory but also in practice, by presenting both implementation methodologies and open source simulation models for two approaches to hybrid DES+SD modelling with Ciw. These contribute

to a collection of open and accessible DES+SD hybrid models for reproduction, modification and scrutinisation; and their implementation is discussed in detail.

In the process of building these models a number of lessons on the nature of this type of integrated hybrid simulation were learned, and these were discussed in Section 5. These include some properties of the limited range of applications to this type of modelling. These also brought to light some limitations of the models. The deterministic nature of the differential equations of the SD components limited analysis on the variability of model outputs. A future research direction could consider using stochastic differential equations instead, possibly allowing for analysis of the interaction between the variabilities of both components. Another potential limitation is that the SD component, despite the parameters being synched regularly with the DES component, is fairly passive, and though not impossible, we could not find an elegant way to trigger DES events based on the SD components, hence the need to solve the differential equations and obtain Equation 9 in order to end the patient's treatment service. It was also discussed that large discrepancies in the time domains or rate or changes between the DES and SD components may introduce some synchronicity issues, although this may also be an indication that hybrid simulation is not an appropriate methodologies for the task.

Similarly there are supposed 'limitations', as discussed in Section 5, that upon further investigation may in fact be inappropriate uses of the integrated hybrid methodology. These include a discord between the important model inputs and the desired model outputs, insufficient interconnectivity between the model components, and insufficient similarities in their effect sizes and time domains. Nevertheless, building and running hybrid simulations may support the modeller in discovering this, and an open and transparent framework, such as that introduced in this paper, would be beneficial.

Acknowledgements

The authors would like to thank Dr Vincent Knight and Professor Paul Harper who provided some valuable feedback on the manuscript.

References

Barbosa, C. and Azevedo, A. Hybrid simulation for complex manufacturing value-chain environments. *Procedia Manufacturing*, 11:1404–1412, 2017.

Benureau, FYC. and Rougier, NP. Re-run, repeat, reproduce, reuse, replicate: transforming code into scientific contributions. *Frontiers in neuroinformatics*, 11:69, 2018.

Brailsford, SC., Eldabi, T., Kunc, M., Mustafee, N., and Osorio, Af. Hybrid simulation modelling in operational research: A state-of-the-art review. *European Journal of Operational Research*, 278(3):721–737, 2019.

Brailsford, SC., Viana, J., Rossiter, S., Channon, AA., and Lotery, AJ. Hybrid simulation for health and social care: the way forward, or more trouble than it's worth? In *2013 Winter Simulations Conference (WSC)*, pages 258–269. IEEE, 2013.

Brito, TB., Trevisan, EFC., and Botter, RC.. A conceptual comparison between discrete and continuous simulation to motivate the hybrid simulation methodology. In *Proceedings of the 2011 Winter Simulation Conference (WSC)*, pages 3910–3922. IEEE, 2011.

Chahal, K. and Eldabi, T. Applicability of hybrid simulation to different modes of governance in uk healthcare. In *2008 Winter Simulation Conference*, pages 1469–1477. IEEE, 2008.

Kloeden, PE., and Pearson, RA. The numerical solution of stochastic differential equations. *The ANZIAM Journal*, 20(1):8–12, 1977.

McKinney, W. Data Structures for Statistical Computing in Python. In Stfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

Monks, T., Currie, CSM., Onggo, BS., Robinson, S., Kunc, M., and Taylor, SJE. Strengthening the reporting of empirical simulation studies: Introducing the stress guidelines. *Journal of Simulation*, 13(1):55–67, 2019.

Morgan, JS., Howick, S. and Belton, V. A toolkit of designs for mixing discrete event simulation and system dynamics. *European Journal of Operational Research*, 257(3):907–918, 2017.

Palmer, GI. *Ciw Documentation*, 2020. `https://ciw.readthedocs.io/en/latest/`.

Palmer, GI., Knight, VA., Harper, PR., and Hawa, AL. Ciw: An open-source discrete event simulation library. *Journal of Simulation*, 13(1):68–82, 2019.

Palmer, GI. and Tian, Y. Source code for ciw hybrid simulations., 2021. `https://zenodo.org/record/4601529`.

Percival, HJW. *Test-driven development with Python*. O'Reilly, 2014.

Pidd, M. *Computer Simulation in Management Science*. Wiley, 2004.

Radhakrishnan, K. and Hindmarsh, A. Description and use of lsode, the livermore solver for ordinary differential equations. 1993.

Ramalho, L. *Fluent python: Clear, concise, and effective programming*, chapter 12. O'Reilly Media, Inc., 2015.

Robinson, S. *September. Simulation: The Practice of Model Development and Use*. Macmillan Education UK. Google-Books-ID: Dtn0oAEACAAJ, 2014.

Sadsad, R., McDonnell, G., Viana, J., Desai, SM., Harper, P., and Brailsford, S. Hybrid modelling case studies. *Discrete-event simulation and system dynamics for management decision making*, pages 295–317, 2014.

Scheidegger, APG., Pereira, TF., de Oliveira, MLM., Banerjee, A., and Montevechi, JAB. An introductory guide for hybrid simulation modelers on the primary simulation methods in industrial engineering identified through a systematic review of the literature. *Computers & Industrial Engineering*, 124:474–492, 2018.

Seeler, KA. *System dynamics: an introduction for mechanical engineers*. Springer, 2014.

Swinerd, C. and McNaught, KR. Design classes for hybrid simulations involving agent-based and system dynamics models. *Simulation Modelling Practice and Theory*, 25:118–133, 2012.

The AnyLogic Company. AnyLogic 8 PLE, 2021. `https://www.anylogic.com/`.

The Ciw library developers. Ciw: 2.1.3, 2020. `http://dx.doi.org/10.5281/zenodo.4068532`.

Tolk, A., Page, EH., and Mittal, S. Hybrid simulation for cyber physical systems: state of the art and a literature review. In *SpringSim (ANSS)*, pages 10–1, 2018.

Uhrmacher, AM., Brailsford, SC., Liu, J., Rabe, M., and Tolk, A. Panelreproducible research in discrete event simulationa must or rather a maybe? In *2016 Winter Simulation Conference (WSC)*, pages 1301–1315. IEEE, 2016.

Viana, J., Brailsford, SC., Harindra, V., and Harper, PR. Combining discrete-event simulation and system dynamics in a healthcare setting: A composite model for chlamydia infection. *European Journal of Operational Research*, 237(1):196–206, 2014.

Virtanen, P., Gommers, R., Oliphant, TE., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, SJ., Brett, M., Wilson, J., Millman, KJ., Mayorov, N., Nelson, ARJ., Jones, E., Kern, R., Larson, E., Carey, CJ., Polat, İ., Feng, Y., Moore, EW., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, EA., Harris, CR., Archibald, AM., Ribeiro, AH., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

Wilson, G., Aruliah, DA., Brown, CT., Chue Hong, NP, Davis, M., Guy, RT., Haddock, SHD., Huff KD., Mitchell, IM., Plumbley, MD., Waugh, B.,. White, EP., Wilson, P. Best practices for scientific computing. *PLoS Biol*, 12(1):e1001745, 2014.

Appendix A. Code to build DES ⊂ SD

This code corresponds to the file `ciw_des_in_sd.py` in Palmer and Tian (2021).

```
import ciw
import numpy as np
from scipy.integrate import odeint

class SD():
    def __init__(self, S, I, M, D, visits_per_day, prop_shop, infection_rate,
                 recovery_rate, prop_immune, rate_lose_immune, death_rate, **kwargs):
        self.S = [S]
        self.I = [I]
        self.M = [M]
        self.D = [D]
        self.visits_per_day = visits_per_day
        self.prop_shop = prop_shop
        self.infection_rate = infection_rate
        self.recovery_rate = recovery_rate
        self.prop_immune = prop_immune
        self.rate_lose_immune = rate_lose_immune
        self.death_rate = death_rate
        self.time = np.array([0])

    def differential_equations(self, y, time_domain, L):
        contact_rate = self.visits_per_day * max(L - 1, 0)
        S, I, M, D = y
        N = S + I + M
        dSdt = (-(S * I * self.prop_shop * contact_rate * self.infection_rate) / (S + (self.prop_shop * I) + M)) + ((1 - self.prop_immune) * self.recovery_rate * I) + (self.rate_lose_immune * M)
        dIdt = ((S * self.prop_shop * I * contact_rate * self.infection_rate) / (S + (self.prop_shop * I) + M)) - ((self.recovery_rate + self.death_rate) * I)
        dMdt = ((self.prop_immune * self.recovery_rate) * I) - (self.rate_lose_immune * M)
        dDdt = self.death_rate * I
        return dSdt, dIdt, dMdt, dDdt

    def solve(self, t, **kwargs):
        L = kwargs['L']
        y0 = (self.S[-1], self.I[-1], self.M[-1], self.D[-1])
        relevant = (self.time_domain <= t) & (self.time_domain >= self.time[-1])
        times_between_events = np.concatenate((np.array([self.time[-1]]), self.time_domain[relevant], np.array([t])), axis=None)
        results = odeint(self.differential_equations, y0, times_between_events, args = (L,))
        S, I, M, D = results.T
        self.S = np.append(self.S, S)
        self.I = np.append(self.I, I)
        self.M = np.append(self.M, M)
        self.D = np.append(self.D, D)
        self.time = np.append(self.time, times_between_events)


```

```
class HybridNode(ciw.Node):
    def release(self, next_individual_index, next_node):
        super().release(next_individual_index, next_node)
        self.simulation.SD.solve(t=self.get_now(), L=self.number_of_individuals)


class HybridSimulation(ciw.Simulation):
    def __init__(self, network, **kwargs):
        self.SD = SD(**kwargs)
        super().__init__(network=network, node_class=HybridNode)

    def simulate_until_max_time(self, max_simulation_time, n_steps):
        self.SD.time_domain = np.linspace(0, max_simulation_time, n_steps)
        super().simulate_until_max_time(max_simulation_time)
        self.SD.solve(t=max_simulation_time, L=self.nodes[1].number_of_individuals)


class SolveSDArrivals(ciw.dists.Distribution):
    def sample(self, t=None, ind=None):
        if t != 0:
            L = self.simulation.nodes[1].number_of_individuals
            self.simulation.SD.solve(t, L=L)
        able_to_shop = self.simulation.SD.S[-1] + self.simulation.SD.M[-1] + (self.simulation.SD.prop_shop
        ↪  * self.simulation.SD.I[-1])
        rate = self.simulation.SD.visits_per_day * able_to_shop
        return ciw.dists.Exponential(rate).sample()
```

Appendix B. Function to run one trial of DES $\subset$ SD

This function is used in the file `des_in_sd.ipynb` in Palmer and Tian (2021).

```
import ciw
import ciw_des_in_sd

def run_trial(trial, number_servers):
    results = {}

    N = ciw.create_network(
        arrival_distributions=[ciw_des_in_sd.SolveSDArrivals()],
        service_distributions=[ciw.dists.Exponential(48)],
        number_of_servers=[number_servers])

    ciw.seed(trial)

    Q = ciw_des_in_sd.HybridSimulation(
        network=N,
        S=200, I=2, M=0, D=0,
        visits_per_day=1/2,
        prop_shop=1/3,
        infection_rate=1,
        recovery_rate=1/14,
        prop_immune=1/4,
        rate_lose_immune=1/75,
        death_rate=1/21)

    Q.simulate_until_max_time(365, 50000)

    results['t'] = Q.SD.time
    results['S'] = Q.SD.S
    results['I'] = Q.SD.I
    results['M'] = Q.SD.M
    results['D'] = Q.SD.D
    return results
```

Appendix C. Code to build SD ⊂ DES

This code corresponds to the file `ciw_sd_in_des.py` in Palmer and Tian (2021).

```
import ciw
import numpy as np
from scipy.integrate import odeint
import random
import math

class SD_Individual(ciw.Individual):
    def __init__(self, id_number, customer_class=0, priority_class=0, simulation=False):
        super().__init__(id_number, customer_class=customer_class, priority_class=priority_class,
        ↪  simulation=simulation)
        self.H = np.array([random.randint(40, 90)])
        self.U = np.array([100 - self.H])
        self.degen_rate = 0.08
        self.cure_rate = 0.4
        self.time = np.array([self.simulation.current_time])

    def differential_equations(self, y, time_domain, in_treatment):
        H, U = y
        if in_treatment:
            dH = ((-self.degen_rate + self.cure_rate) * U * H) / (H + U)
            dU = ((self.degen_rate - self.cure_rate) * U * H) / (H + U)
        else:
            dH = -(self.degen_rate * U * H) / (H + U)
            dU = (self.degen_rate * U * H) / (H + U)
        return dH, dU

    def solve(self, t, **kwargs):
        y0 = self.H[-1], self.U[-1]
        in_treatment = kwargs['in_treatment']
        relevant = (self.simulation.time_domain <= t) & (self.simulation.time_domain >= self.time[-1])
        times_between_events = np.concatenate(
            (np.array([self.time[-1]]), self.simulation.time_domain[relevant], np.array([t])),
            axis=None)
        results = odeint(
            self.differential_equations, y0,
            times_between_events, args=(in_treatment,))
        H, U = results.T
        self.H = np.append(self.H, H)
        self.U = np.append(self.U, U)
        self.time = np.append(self.time, times_between_events)


class Treatment(ciw.dists.Distribution):
    def sample(self, t, ind):
        ind.solve(t, in_treatment=False)
        if ind.H[-1] < 10:
            ind.status = 'Untreatable'
            return 0
        ind.status = 'Treated'
        return math.log(ind.H[-1]/(99*ind.U[-1])) / (ind.degen_rate - ind.cure_rate)


class HybridNode(ciw.Node):
    def detatch_server(self, server, individual):
        super().detatch_server(server, individual)
        individual.solve(self.get_now(), in_treatment=True)


class HybridSimulation(ciw.Simulation):
    def __init__(self, network,
                 deadlock_detector=ciw.deadlock.NoDetection(),
                 node_class=HybridNode,
```

```
                  individual_class=SD_Individual):
        super().__init__(network=network,
                         node_class=HybridNode,
                         individual_class=SD_Individual)

    def simulate_until_max_time(self, max_simulation_time, n_steps):
        self.time_domain = np.linspace(0, max_simulation_time, n_steps)
        super().simulate_until_max_time(max_simulation_time)
```

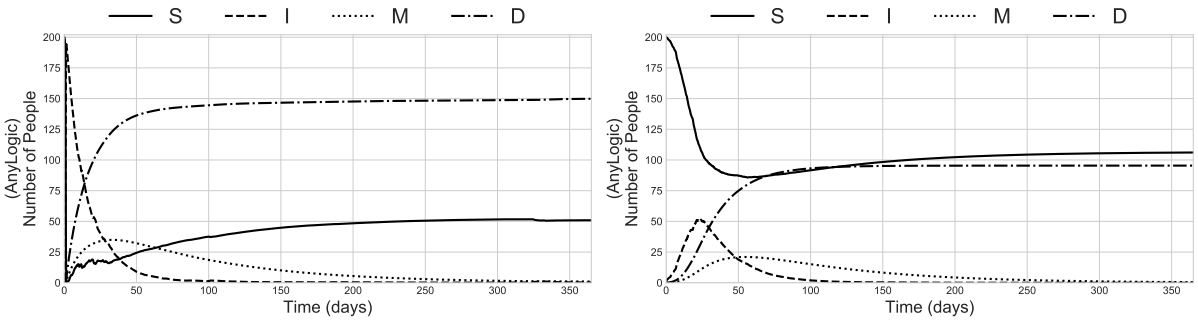
Appendix D. Function to run one trial of SD ⊂ DES

This function is used in the file `sd_in_des.ipynb` in Palmer and Tian (2021).

```
import ciw
import ciw_sd_in_des

def run_trial(trial, number_servers):
    results = {}

    N = ciw.create_network(
        arrival_distributions=[ciw.dists.Exponential(2)],
        service_distributions=[ciw_sd_in_des.Treatment()],
        number_of_servers=[number_servers])

    ciw.seed(trial)

    Q = ciw_sd_in_des.HybridSimulation(network=N)
    Q.simulate_until_max_time(365*3, 50000)
    recs = pd.DataFrame(Q.get_all_records())
    success_rate = (recs[(recs['arrival_date'] >= 365) & (recs['arrival_date'] <= 2*365)]['service_time']
    ↪  == 0).mean()

    return success_rate
```

Appendix E. Results in AnyLogic

Both models presented in this paper were also built in AnyLogic (The AnyLogic Company (2021)), and the AnyLogic models are archived at Palmer and Tian (2021). Here are the equivalent figures to Figures 7 and 9 with data produced by the AnyLogic models.



(a) Maximum capacity of one.

(b) Maximum capacity of three.

Figure E1. Equivalent plots to Figure 7 with data produced by an AnyLogic model.

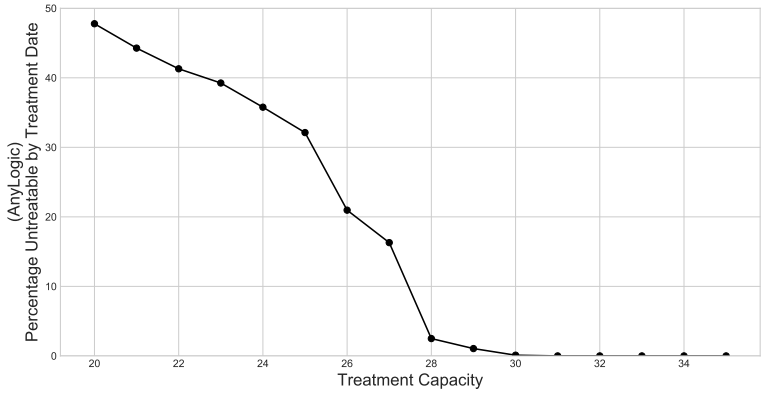


Figure E2. Equivalent plot to Figure 9 with data produced by an AnyLogic model.